Tutorial do PostgreSQL 7.3.2

The PostgreSQL Global Development Group

Índice

Prefácio

1. O que é o PostgreSQL?

O PostgreSQL é um sistema de gerenciamento de banco de dados objeto-relacional (SGBDOR) baseado no POSTGRES, Versão 4.2[1], desenvolvido no Departamento de Ciência da Computação da Universidade da Califórnia em Berkeley. O projeto POSTGRES, liderado pelo Professor Michael Stonebraker, foi patrocinado pelas seguintes instituições: Defense Advanced Research Projects Agency (DARPA); Army Research Office (ARO); National Science Foundation (NSF); e ESL, Inc.

O PostgreSQL descende deste código original de Berkeley, possuindo o código fonte aberto. Fornece suporte às linguagens SQL92/SQL99 além de outras funcionalidades modernas.

O POSTGRES foi o pioneiro em muitos conceitos objeto-relationais que agora estão se tornando disponíveis em alguns bancos de dados comerciais. Os Sistemas de Gerenciamento de Bancos de Dados Relacionais (SGBDR) tradicionais suportam um modelo de dados que consiste em uma coleção de relações com nome, contendo atributos de um tipo específico. Nos sistemas comerciais em uso, os tipos possíveis incluem número de ponto flutuante, inteiro, cadeia de caracteres, monetário e data. É largamente reconhecido que este modelo não é adequado para aplicações futuras de processamento de dados. O modelo relacional substituiu com sucesso os modelos anteriores em parte devido à sua “simplicidade Espartana”. Entretanto, esta simplicidade tornou a implementação de certas aplicações muito difícil. O PostgreSQL oferece um substancial poder adicional, devido à incorporação dos conceitos mostrados abaixo de uma forma que os usuários podem facilmente estender o sistema:

- herança
- tipos de dados
- funções

Outras funcionalidades fornecem poder e flexibilidade adicionais:

- restrições
- gatilhos
- regras
- integridade da transação

Estas funcionalidades colocam o PostgreSQL dentro da categoria de bancos de dados referida como *objeto-relacional*. Repare que isto é diferente daqueles referidos como *orientados a objetos* que, em geral, não são muito adequados para dar suporte às linguagens de banco de dados relacionais tradicionais. Portanto, embora o PostgreSQL possua algumas funcionalidades de orientação a objetos, está firmemente ligado ao mundo dos bancos de dados relacionais. Na verdade, alguns bancos de dados comerciais incorporaram recentemente funcionalidades nas quais o PostgreSQL foi o pioneiro.

2. Uma breve história do PostgreSQL

O sistema de gerenciamento de banco de dados objeto-relacional hoje em dia conhecido por PostgreSQL (e por um breve período de tempo chamado Postgres95) é derivado do pacote POSTGRES escrito na Universidade da Califórnia em Berkeley. Com mais de uma década de desenvolvimento por trás, o PostgreSQL é o mais avançado

1. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html

banco de dados de código aberto disponível em qualquer lugar, oferecendo controle de concorrência multi-versão, suportando praticamente todas as construções do SQL (incluindo subconsultas, transações, tipos definidos pelo usuário e funções), e dispondo de um amplo conjunto de ligações com linguagens procedurais (incluindo C, C++, Java, Perl, Tcl e Python).

2.1. O projeto POSTGRES de Berkeley

A implementação do SGBD POSTGRES começou em 1986. Os conceitos iniciais para o sistema foram apresentados em *The design of POSTGRES*, e a definição do modelo de dados inicial apareceu em *The POSTGRES data model*. O projeto do sistema de regras nesta época foi descrito em *The design of the POSTGRES rules system*. Os princípios básicos e a arquitetura do gerenciador de armazenamento foram detalhados em *The design of the POSTGRES storage system*.

O Postgres passou por várias versões desde então. A primeira “versão de demonstração” do sistema se tornou operacional em 1987, e foi exibida em 1988 na Conferência ACM-SIGMOD. A versão 1, descrita em *The implementation of POSTGRES*, foi liberada para alguns poucos usuários externos em junho de 1989. Em resposta à crítica ao primeiro sistema de regras (*A commentary on the POSTGRES rules system*), o sistema de regras foi re-projetado (*On Rules, Procedures, Caching and Views in Database Systems*) e a versão 2 foi liberada em junho de 1990, contendo um novo sistema de regras. A versão 3 surgiu em 1991 adicionando suporte a múltiplos gerenciadores de armazenamento, um executor de consultas melhorado, e um sistema de regras de reescrita reescrito. Para a maior parte, as versões seguintes até o Postgres95 (veja abaixo) focaram a portabilidade e a confiabilidade.

O POSTGRES tem sido usado para implementar muitas aplicações diferentes de pesquisa e produção, incluindo: sistema de análise de dados financeiros, pacote de monitoramento de desempenho de turbina a jato, banco de dados de acompanhamento de asteróides, banco de dados de informações médicas, além de vários sistemas de informações geográficas. O POSTGRES também tem sido usado como uma ferramenta educacional em diversas universidades. Finalmente, a Illustra Information Technologies (posteriormente incorporada pela Informix[2], que agora pertence à IBM[3]) pegou o código e comercializou. O POSTGRES se tornou o gerenciador de dados primário para o projeto de computação científica Sequoia 2000[4] nos fins de 1992.

O tamanho da comunidade de usuários externos praticamente dobrou durante o ano de 1993. Começou a se tornar cada vez mais óbvio que a manutenção do código do protótipo e seu suporte estava consumindo uma grande quantidade de tempo que deveria ser dedicado a pesquisas sobre bancos de dados. Em um esforço para reduzir esta sobrecarga de suporte, o projeto do POSTGRES de Berkeley terminou oficialmente com a versão 4.2.

2.2. O Postgres95

Em 1994, Andrew Yu e Jolly Chen adicionaram um interpretador da linguagem SQL ao POSTGRES. O Postgres95 foi em seguida liberado para a Web para encontrar seu caminho no mundo como um descendente de código aberto do código original do POSTGRES de Berkeley.

O código do Postgres95 era totalmente escrito em ANSI C e reduzido em tamanho em 25%. Muitas mudanças internas melhoraram o desempenho e a facilidade de manutenção. O Postgres95 release 1.0.x era 30-50% mais rápido em comparação com o POSTGRES versão 4.2, utilizando o Wisconsin Benchmark. Além da correção de erros, as melhorias principais foram as seguintes:

- A linguagem de consultas PostQUEL foi substituída pelo SQL (implementado no servidor). As subconsultas não foram suportadas até o PostgreSQL (veja abaixo), mas podiam ser imitadas no Postgres95 através de funções SQL definidas pelo usuário. As agregações foram re-implementadas. O suporte para a cláusula GROUP

2. http://www.informix.com/
3. http://www.ibm.com/
4. http://meteora.ucsd.edu/s2k/s2k_home.html

BY das consultas também foi adicionado. A interface da `libpq` permaneceu disponível para os programas escritos na linguagem C.

- Adicionalmente ao programa monitor, um novo programa (psql) foi fornecido para consultas SQL interativas utilizando o Readline do GNU.
- Uma nova biblioteca cliente, `libpgtcl`, suportava clientes baseados no Tcl. Uma shell, `pgtclsh`, permitia que os novos comandos Tcl fizessem uma interface entre os programas Tcl e o servidor Postgres95.
- A interface para objetos grandes foi refeita. A Inversão de objetos grandes era o único mecanismo para armazená-los (O sistema de arquivos Inversão foi removido).
- O sistema de regras no nível de instância foi removido. As regras ainda estavam disponíveis através de regras de reescrita.
- Um breve tutorial introduzindo as funcionalidades regulares da linguagem SQL, assim como as do Postgres95, foi distribuído junto com o código fonte.
- O utilitário make do GNU (no lugar do make do BSD) foi utilizado para a geração. Além disso, o Postgres95 podia ser compilado com o GCC sem correções (alinhamento de dados para a precisão dupla foi corrigido).

2.3. O PostgreSQL

Em 1996 se tornou claro que o nome “Postgres95” não resistiria ao teste do tempo. Foi escolhido então um novo nome, PostgreSQL, para refletir o relacionamento entre o POSTGRES original e as versões mais recentes com funcionalidade SQL. Ao mesmo tempo, foi mudado o número da versão para começar em 6.0, colocando os números de volta na seqüência original começada pelo projeto POSTGRES de Berkeley.

A ênfase durante o desenvolvimento do Postgres95 era identificar e compreender os problemas existentes no código do servidor. Com o PostgreSQL, a ênfase foi mudada para a melhoria das funcionalidades e dos recursos, embora o trabalho continuasse em todas as áreas.

As principais melhorias no PostgreSQL incluem:

- O bloqueio no nível de tabela foi substituído por um sistema de concorrência multi-versão, que permite aos que estão lendo continuarem a ler dados consistentes durante a atividade de escrita, e permite efetuar cópias de segurança através do pg_dump enquanto o banco de dados se mantém disponível para consultas.
- A implementação de funcionalidades importantes no servidor, incluindo subconsultas, padrões, restrições e gatilhos.
- A incorporação de funcionalidades adicionais compatíveis com a linguagem SQL92, incluindo chaves primárias, identificadores entre aspas, conversão implícita de tipo de cadeias de caracteres literais, conversão explícita de tipos e inteiros binários e hexadecimais.
- Os tipos nativos foram aperfeiçoados, incluindo uma grande variedade de tipos para data e hora e suporte adicional para tipo geométrico.
- A velocidade geral do código do servidor foi melhorada em aproximadamente 20-40%, e o tempo de inicialização do servidor foi reduzido em 80% desde que a versão 6.0 foi liberada.

3. O que está contido neste manual

Seja bem-vindo ao PostgreSQL e ao *Tutorial do PostgreSQL*. Os poucos capítulos que se seguem têm por objetivo fornecer uma breve introdução ao PostgreSQL, aos conceitos de banco de dados relacional e à linguagem SQL, para os que são iniciantes em qualquer um destes tópicos. Somente assume-se um conhecimento geral sobre a utilização de computadores. Nenhuma experiência particular com Unix ou em programação é requerida. Este tutorial tem por objetivo principal fornecer experiência prática com relação aos aspectos importantes do PostgreSQL. Não existe nenhuma intenção em se dar um tratamento completo ou abrangente dos tópicos cobertos.

Após ler este tutorial pode-se prosseguir através da leitura do *Guia do Usuário do PostgreSQL* para obter mais conhecimento formal sobre a linguagem SQL, ou o *Guia do Programador do PostgreSQL* para obter informações sobre o desenvolvimento de aplicações para o PostgreSQL. Os que instalam e gerenciam seus próprios servidores também devem ler o *Guia do Administrador do PostgreSQL*.

4. Revisão dos recursos de documentação

A documentação do PostgreSQL está organizada em diversos livros:

Tutorial do PostgreSQL

Uma introdução informal para os novos usuários.

Guia do Usuário do PostgreSQL

Documenta o ambiente da linguagem de consulta SQL, incluindo tipos de dados e funções, assim como o ajuste do desempenho no nível de usuário. Todo usuário do PostgreSQL deve lê-lo.

Guia do Administrador do PostgreSQL

Informações sobre a instalação e a administração do servidor. Todos os que administram um servidor PostgreSQL, tanto para uso pessoal quanto para uso por outras pessoas, devem lê-lo.

Guia do Programador do PostgreSQL

Informações avançadas para programadores de aplicativos. Os tópicos incluem a extensibilidade de tipo e de função, interfaces com bibliotecas e questões relativas ao projeto de aplicativos.

Manual de Referência do PostgreSQL

Páginas de referência contendo a sintaxe para os comandos SQL, programas para a estação cliente e servidor. Este manual é um auxílio aos Guias do Usuário, do Administrador e do Programador.

Guia do Desenvolvedor do PostgreSQL

Informações para os desenvolvedores do PostgreSQL. Feito para àqueles que estão contribuindo para o projeto PostgreSQL. As informações sobre o desenvolvimento de aplicativos estão mostradas no *Guia do Programador*.

Além deste conjunto de manuais, existem outros recursos para ajudar na instalação e uso do PostgreSQL:

man pages

As páginas do *Manual de Referência* no formato tradicional “man” do Unix. Não existe diferença no conteúdo.

FAQs

As listas de perguntas freqüentemente formuladas (Frequently Asked Questions - FAQ) documentam tanto questões gerais como algumas questões específicas para uma plataforma.

READMEs

Os arquivos README (leia-me) estão disponíveis em alguns pacotes contribuídos.

Sítio na Web

O sítio na Web do PostgreSQL [5] possui detalhes sobre sobre a última versão, funcionalidades a serem incorporadas, e outras informações para tornar o trabalho ou a diversão com o PostgreSQL mais produtiva.

Listas de discussão

As listas de discussão são um bom lugar para se conseguir ter as perguntas respondidas, para trocar experiência com outros usuários, e para fazer contato com os desenvolvedores. Consulte a seção User's Lounge[6] do sítio na Web do PostgreSQL para obter mais informações.

Você mesmo!

O PostgreSQL é um trabalho de código aberto e, por isso, depende do apoio permanente da comunidade de usuários. Quando começamos a utilizar o PostgreSQL dependemos da ajuda de outros, tanto através da documentação quanto das listas de discussão. Considere então em retribuir seus conhecimentos. Se você aprender alguma coisa que não faz parte da documentação, escreva e contribua. Se você adicionar funcionalidades ao código, contribua com estas funcionalidades.

Mesmo àqueles que não possuem muita experiência podem fazer correções e pequenas alterações na documentação, e esta é uma boa maneira de se começar. A lista de discussão <`pgsql-docs@postgresql.org`> é um lugar para se visitar com freqüência (N.T. - No Brasil veja PostgreSQL Brasil[7]).

5. Terminologia e notação

Um *administrador* normalmente é a pessoa responsável pela instalação e funcionamento do servidor. Um *usuário* pode ser qualquer um usando, ou querendo usar, qualquer parte do sistema PostgreSQL. Estes termos não devem ser interpretados ao pé da letra; este conjunto de documentos não estabelece premissas sobre os procedimentos do administrador do sistema.

É utilizado `/usr/local/pgsql/` como sendo o diretório raiz da instalação e `/usr/local/pgsql/data` como o diretório contendo os arquivos dos banco de dados. Estes diretórios podem ser outros em sua máquina, os detalhes podem ser vistos no *Guia do Administrador* (N.T. `/var/lib/pgsql/` e `/var/lib/pgsql/data` no Mandrake Linux).

Na sinopse dos comandos os colchetes (`[]`) indicam uma frase ou palavra chave opcional. Qualquer coisa entre chaves (`{}`) contendo, também, barras verticais (`|`), indica que uma das alternativas deve ser escolhida.

Os exemplos mostram comandos executados a partir de várias contas e programas. Os comandos executados a partir de uma "shell" do Unix podem estar precedidos por um caractere de cifrão ("`$`"). Os comandos executados a partir da conta de um usuário, tal como root ou postgres, são especialmente sinalizados e explicados. Os comandos SQL podem estar precedidos por "=>" ou não ter nada os precedendo, dependendo do contexto.

5. http://www.postgresql.org
6. http://www.postgresql.org/users-lounge/
7. http://br.groups.yahoo.com/group/postgresql-br/

> **Nota:** A notação para sinalizadores de comandos não é totalmente consistente através do conjunto de documentos. Por favor relate os problemas encontrados para a lista de discussão da documentação em <pgsql-docs@postgresql.org>.

6. Guia para relatar erros

Quando é encontrado algum erro no PostgreSQL nós desejamos conhecê-lo. Seus relatórios de erros são uma parte importante para tornar o PostgreSQL mais confiável, porque mesmo o mais extremo cuidado não pode garantir que todas as partes do PostgreSQL vão funcionar em todas as plataformas sob qualquer circunstância.

As sugestões abaixo têm por objetivo ajudá-lo a preparar seu relatório de erro, para que este possa ser tratado de uma maneira efetiva. Ninguém é obrigado a segui-las, mas são feitas para ser uma vantagem para todos.

Nós não podemos prometer corrigir todos os erros imediatamente. Se o erro for óbvio, crítico, ou afetar muitos usuários, existe uma boa chance de que alguém vai olhá-lo. Pode acontecer, também, nós solicitarmos a atualização para uma nova versão, para ver se o erro também acontece na nova versão. Também podemos decidir que o erro não poderá ser corrigido antes de uma grande reescrita que planejamos fazer. Ou, talvez, é simplesmente muito difícil corrigi-lo e existem coisas mais importantes na agenda. Se for necessária ajuda imediata, deve ser levada em consideração a contratação de um suporte comercial.

6.1. Identificando erros

Antes de relatar um erro, por favor leia e releia a documentação para verificar se realmente pode ser feito o que está se tentando fazer. Se não está claro na documentação que algo pode ou não ser feito, por favor informe isto também; é um erro da documentação. Se acontecer do programa fazer algo diferente do que está escrito na documentação, isto também é um erro. Pode incluir, mas não se restringe, às seguintes circunstâncias:

- O programa termina com um erro fatal, ou com uma mensagem de erro do sistema operacional que aponta para um erro no programa (um exemplo oposto seria uma mensagem de “disco cheio”, uma vez que o próprio usuário deve corrigir este problema).
- O programa produz uma saída errada para uma entrada específica.
- O programa se recusa a aceitar uma entrada válida (conforme definido na documentação).
- O programa aceita uma entrada inválida sem enviar uma mensagem de erro. Porém, tenha em mente que a sua idéia de entrada errada pode ser a nossa idéia de uma extensão, ou a compatibilidade com a prática tradicional.
- O PostgreSQL falha ao compilar, montar ou instalar de acordo com as instruções, em uma plataforma suportada.

Aqui “programa” se refere a qualquer executável, e não apenas ao processo servidor.

Estar lento ou consumir muitos recursos não é necessariamente um erro. Leia a documentação ou faça perguntas em alguma lista de discussão para pedir ajuda para ajustar seus aplicativos. Não agir de acordo com o padrão SQL também não é necessariamente um erro, a menos que a aderência com a funcionalidade específica esteja explicitamente declarada.

Antes de prosseguir, verifique a lista TODO (a fazer) e a FAQ para ver se o seu erro já não é conhecido. Se você não conseguir decodificar a informação da lista TODO, relate seu problema. O mínimo que podemos fazer é tornar a lista TODO mais clara.

6.2. O que relatar

A coisa mais importante a ser lembrada quando vamos relatar erros é declarar todos os fatos, e somente os fatos. Não especule sobre o que você acha que está certo ou errado, o que "parece que deva ser feito", ou qual parte do programa está falhando. Se você não está familiarizado com a implementação você provavelmente vai supor errado, e não vai nos ajudar nem um pouco. E, mesmo que você esteja familiarizado, uma explanação educada é um grande suplemento, mas não substitui os fatos. Se nós formos corrigir o erro, nós temos que vê-lo acontecer primeiro. Informar fatos cruamente é relativamente direto (provavelmente pode ser copiado e colado a partir da tela), mas geralmente detalhes importantes são deixados de fora porque alguém pensou que não tinha importância, ou que o relatório seria entendido de qualquer maneira.

Os seguintes itens devem estar contidos em todo relatório de erro:

- A seqüência exata dos passos, *desde o início do programa*, necessários para reproduzir o problema. Isto deve estar autocontido; não é suficiente enviar apenas um comando SELECT sem enviar os comandos CREATE TABLE e INSERT que os precederam, se a saída depender dos dados contidos nas tabelas. Nós não temos tempo para realizar a engenharia reversa do esquema do seu banco de dados e, se nós tivermos que criar os nossos próprios dados, nós provavelmente não vamos conseguir reproduzir o problema. O melhor formato para realizar um caso de teste, para problemas relacionados com a linguagem de consulta, é um arquivo que possa ser executado a partir do aplicativo psql e que mostre o problema (certifique-se de que não exista nada em seu arquivo de inicialização `~/.psqlrc`). Uma maneira fácil de começar este arquivo é usar o pg_dump para gerar as declarações da tabela e dos dados necessários para montar o cenário, e depois incluir a consulta problemática. Encorajamos você a minimizar o tamanho de seu exemplo, mas isto não é absolutamente necessário. Se o erro for reproduzível, nós o encontraremos de qualquer forma.

  Se o seu aplicativo utiliza alguma outra interface no cliente tal como o PHP então, por favor, tente isolar a consulta problemática. Provavelmente nós não vamos configurar um servidor Web para reproduzir o seu problema. De qualquer maneira lembre-se de fornecer os arquivos de entrada exatos, e não suponha que o problema aconteça com "arquivos grandes" ou "bancos de dados de tamanho médio", etc... uma vez que estas informações são muito imprecisas para que possam ser utilizadas.

- A saída produzida. Por favor, não diga que "não funcionou" ou que "deu pau". Se houver uma mensagem de erro mostre-a, mesmo que você não consiga entendê-la. Se o programa terminar com um erro do sistema operacional, diga qual. Se nada acontecer, informe. Mesmo que o resultado do seu caso de teste seja o término anormal do programa, ou seja óbvio de alguma outra maneira, pode ser que isto não aconteça em nossa plataforma. A coisa mais fácil a ser feita é copiar a saída do terminal, se for possível.

  > **Nota:** No caso de erros fatais a mensagem de erro informada pelo cliente pode não conter toda a informação disponível. Por favor, olhe também o `log` produzido no servidor de banco de dados. Se você não mantém a saída do `log` do servidor, esta é uma boa hora para começar a fazê-lo.

- A saída esperada é uma informação importante a ser declarada. Se você escreve apenas "Este comando produz esta saída." ou "Isto não é o que eu esperava.", nós podemos executar, olhar a saída, e achar que está tudo correto e é exatamente o que nós esperávamos que fosse. Nós não temos que perder tempo para decodificar a semântica exata por trás de seus comandos. Abstenha-se especialmente de dizer meramente "Isto não é o que o SQL diz ou o que o Oracle faz". Pesquisar o comportamento correto do SQL não é uma tarefa divertida, nem nós sabemos como todos os outros bancos de dados relacionais existentes se comportam (se o seu problema é o término anormal de seu programa, este item obviamente pode ser omitido).
- Qualquer opção de linha de comando e outras opções de inicialização, incluindo as variáveis de ambiente relacionadas ou arquivos de configuração que foram mudados com relação ao padrão. Novamente, seja exato.

Se estiver sendo utilizada uma distribuição pré-configurada que inicializa o servidor de banco de dados durante o `boot`, deve-se tentar descobrir como isto é feito.

- Qualquer coisa feita que seja diferente das instruções de instalação.
- A versão do PostgreSQL. Pode-se executar o comando `SELECT version();` para descobrir a versão do servidor ao qual se está conectado. A maioria dos programas executáveis suporta a opção `--version`; ao menos `postmaster --version` e `psql --version` devem funcionar. Se a função ou as opções não existirem, então a versão sendo usada é muito antiga e merece ser atualizada. Também pode ser visto o arquivo `README` no diretório fonte, ou o arquivo com o nome da distribuição ou o nome do pacote. Se a versão for pré-configurada, como RPMs, informe, incluindo qualquer sub-versão que o pacote possa ter. Se estiver se referindo a uma versão do CVS isto deve ser mencionado, incluindo a data e a hora.

  Se a sua versão for anterior a 7.3.2 provavelmente nós lhe pediremos que atualize. Existem toneladas de correções de erro a cada nova liberação, sendo este o motivo das novas liberações.

- Informações da plataforma. Isto inclui o nome do núcleo e a versão, a biblioteca C, o processador e a memória. Na maioria dos casos é suficiente informar o fornecedor e a versão, mas não se deve assumir que todo mundo sabe exatamente o que o “Debian” contém, ou que todo mundo use Pentium. Se houver problemas de instalação então as informações sobre compilador, `make`, etc... também são necessárias.

Não tenha medo que seu relatório de erro se torne muito longo. Este é um fato da vida. É melhor relatar tudo da primeira vez do que nós termos que extrair os fatos de você. Por outro lado, se os seus arquivos de entrada são enormes, é justo perguntar primeiro se alguém está interessado em vê-los.

Não perca todo o seu tempo tentando descobrir que mudanças na sua entrada fazem o problema desaparecer. Isto provavelmente não vai ajudar a resolver o problema. Se for visto que o erro não pode ser corrigido imediatamente, você ainda vai ter tempo para compartilhar sua descoberta com os outros. Também, novamente, não perca seu tempo advinhando porque o erro existe. Nós o encontraremos brevemente.

Ao escrever o relatório de erro, por favor escolha uma terminologia que não confunda. O pacote de software em seu todo é chamado de “PostgreSQL” e, algumas vezes, de “Postgres” para abreviar. Se estiver se referindo especificamente ao processo servidor mencione isto, não diga apenas que o “PostgreSQL caiu”. A queda de um único processo do servidor é bem diferente da queda do processo “postmaster” pai; por favor não diga “o postmaster caiu” quando um único processo servidor caiu, nem o contrário. Além disso os programas cliente, como o terminal interativo “psql”, são completamente separados do servidor. Por favor, tente especificar se o problema está no lado do cliente ou no lado do servidor.

6.3. Aonde relatar os erros

De uma maneira geral os relatórios de erro devem ser enviados para a lista de discussão de relatórios de erros em <`pgsql-bugs@postgresql.org`>. É necessária a utilização de um assunto descritivo para a mensagem de correio eletrônico, talvez uma parte da própria mensagem de erro.

Outro método é preencher o relatório de erro disponível no sítio do projeto na Web em http://www.postgresql.org/. O preenchimento desta forma faz com que este seja enviado para a lista de discussão <`pgsql-bugs@postgresql.org`>.

Não envie o relatório de erro para nenhuma lista de discussão dos usuários, tal como <`pgsql-sql@postgresql.org`> ou <`pgsql-general@postgresql.org`>. Estas listas de discussão são para responder perguntas dos usuários, e seus subscritores normalmente não desejam receber relatórios de erro. Mais importante ainda, eles provavelmente não vão conseguir corrigir o erro.

Por favor, também *não* envie relatórios para a lista de discussão dos desenvolvedores em <`pgsql-hackers@postgresql.org`>. Esta lista é para discutir o desenvolvimento do PostgreSQL e nós

gostamos de manter os relatórios de erro em separado. Nós podemos decidir discutir o seu relatório de erro em `pgsql-hackers`, se o problema necessitar uma maior averiguação.

Se você tiver problema com a documentação, o melhor lugar para relatar é na lista de discussão da documentação em <`pgsql-docs@postgresql.org`>. Por favor, seja específico sobre qual parte da documentação você está descontente. (N.T. Informe os erros encontrados na tradução deste manual a <`halleypo@yahoo.com.br`>)

Se o seu erro for algum problema de portabilidade ou uma plataforma não suportada, envie uma mensagem de correio eletrônico para <`pgsql-ports@postgresql.org`>, para que nós (e você) possamos trabalhar para portar o PostgreSQL para esta plataforma.

> **Nota:** Devido à grande quantidade de `spam` na Internet, todos os endereços de correio eletrônico acima são de listas de discussão fechadas. Ou seja, você precisa subscrever a lista primeiro para depois poder enviar mensagens (entretanto, você não precisa subscrever para utilizar o formulário de relatório de erro da Web). Se você deseja enviar uma mensagem de correio eletrônico, mas não deseja receber o tráfego da lista, você pode subscrever e configurar sua opção de subscrição com `nomail`. Para maiores informações envie uma mensagem para <`majordomo@postgresql.org`> contendo apenas a palavra `help` no corpo da mensagem.

Capítulo 1. Iniciando

1.1. Instalação

Antes de poder usar o PostgreSQL este deve estar instalado, é claro. É possível que o PostgreSQL já esteja instalado em sua máquina, seja porque está incluído na distribuição do sistema operacional, ou porque o administrador do sistema fez a instalação. Se este for o caso, deve-se obter informações na documentação do sistema operacional ou com o administrador do sistema sobre como acessar o PostgreSQL.

Se você não tiver certeza que o PostgreSQL está disponível, ou que pode ser usado para os seus experimentos, você poderá fazer a instalação por si mesmo. Proceder desta maneira não é difícil e pode ser um bom exercício. O PostgreSQL pode ser instalado por qualquer usuário sem privilégios, porque nenhum acesso de super-usuário (root) é necessário.

Se você for instalar o PostgreSQL por si mesmo, então consulte o *Guia do Administrador do PostgreSQL* para ler as instruções de instalação, e depois retorne para este manual quando a instalação estiver completa. Certifique-se de seguir de perto a seção relativa à configuração das variáveis de ambiente apropriadas.

Se o administrador do sistema não fez a configuração conforme o padrão, talvez algum trabalho adicional deva ser feito. Por exemplo, se a máquina servidora de banco de dados for uma máquina remota, será necessário definir a variável de ambiente `PGHOST` com o nome da máquina servidora de banco de dados. A variável de ambiente `PGPORT` talvez também tenha que ser definida. A regra básica é esta: se você tentar iniciar um programa aplicativo e este informar que não pode se conectar ao banco de dados, você deve consultar o administrador do servidor ou, caso seja você mesmo, a documentação para ter certeza de que o seu ambiente está corretamente configurado. Se você não entendeu o parágrafo anterior então, por favor, leia a próxima seção.

1.2. Fundamentos da Arquitetura

Antes de prosseguirmos, você deve entender as bases da arquitetura de sistema do PostgreSQL. Compreendendo como as partes do PostgreSQL interagem entre si torna este capítulo mais claro.

No jargão de banco de dados, o PostgreSQL utiliza o modelo cliente-servidor. Uma sessão do PostgreSQL consiste dos seguintes processos (programas) cooperando entre si:

- Um processo servidor, o qual gerencia os arquivos do banco de dados, aceita as conexões dos aplicativos cliente com o banco de dados, e executa as ações no banco de dados em nome dos clientes. O programa servidor de banco de dados chama-se `postmaster`.
- O aplicativo cliente do usuário (frontend) que deseja executar operações de banco de dados. Os aplicativos cliente podem ter naturezas muito diversas: o cliente pode ser uma ferramenta no modo caractere, um aplicativo gráfico, um servidor Web que acessa o banco de dados para mostrar páginas da Web, ou uma ferramenta especializada para manutenção do banco de dados. Alguns aplicativos cliente são fornecidos junto com a distribuição do PostgreSQL, sendo a maioria desenvolvida pelos usuários.

É típico em aplicativos cliente-servidor o cliente e o servidor estarem em máquinas diferentes. Neste caso eles se comunicam através de uma conexão de rede TCP/IP. Deve-se ter isto em mente, porque arquivos que podem ser acessados na máquina cliente podem não ser acessíveis (ou podem somente ser acessíveis usando um nome de arquivo diferente) pela máquina servidora.

O servidor PostgreSQL pode tratar múltiplas conexões concorrentes dos clientes. Para esta finalidade é iniciado (“fork”) um novo processo para cada conexão. Deste ponto em diante, o cliente e o novo processo servidor passam

a se comunicar sem a intervenção do processo original `postmaster`. Portanto, o `postmaster` está sempre executando aguardando por novas conexões dos clientes, enquanto os processos servidores associados aos clientes surgem e desaparecem (tudo isso, obviamente, é invisível para o usuário sendo somente mencionado para ficar completo).

1.3. Criando um banco de dados

O primeiro teste a ser feito para ver se é possível acessar o servidor de banco de dados é tentar criar um banco de dados. Um servidor PostgreSQL pode gerenciar muitos bancos de dados. Tipicamente, um banco de dados em separado é usado para cada projeto ou para cada usuário.

Possivelmente o administrador já criou um banco de dados para o seu uso. Ele deve ter dito qual é o nome de seu banco de dados. Neste caso esta etapa pode ser pulada, indo-se direto para a próxima seção.

para criar um novo banco de dados chamado `meu_bd` deve ser usado o seguinte comando:

```
$ createdb meu_bd
```

O que deve produzir a resposta:

```
CREATE DATABASE
```

Se isto acontecer, esta etapa foi executada com sucesso e o resto da seção pode ser saltada.

Se aparecer uma mensagem parecida com

```
createdb: command not found
```

então o PostgreSQL não foi instalado adequadamente. Ou não foi instalado, ou o caminho de procura não está correto. Tente executar o comando utilizando o caminho absoluto:

```
$ /usr/local/pgsql/bin/createdb meu_bd
```

O caminho na sua máquina pode ser diferente (N.T. /usr/bin/createdb no Mandrake Linux). Faça contato com o administrador ou verifique novamente as instruções de instalação para corrigir a situação.

Outra resposta pode ser esta:

```
psql: could not connect to server: Connection refused
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
createdb: database creation failed
```

Isto indica que o servidor não foi inicializado, ou que não foi inicializado aonde o `createdb` esperava que estivesse. Novamente, verifique as instruções de instalação ou consulte o administrador.

Se você não possuir o privilégio necessário para criar bancos de dados, a seguinte mensagem será exibida:

```
ERROR:  CREATE DATABASE: permission denied
createdb: database creation failed
```

Nem todo usuário possui autorização para criar novos bancos de dados. Se o PostgreSQL recusar a criação do banco de dados, então o administrador deve conceder a permissão para criar bancos de dados para você. Consulte

o administrador caso isto ocorra. Se você instalou o PostgreSQL por si mesmo, então você deve se logar usando a mesma conta utilizada para inicializar o servidor para poder usar este tutorial. [1]

Você também pode criar bancos de dados com outros nomes. O PostgreSQL permite a criação de qualquer número de bancos de dados em uma instalação. Os nomes dos bancos de dados devem ter um primeiro caractere alfabético, estando limitados a um comprimento de 63 caracteres. Uma escolha conveniente é criar o banco de dados com o mesmo nome de seu usuário corrente. Muitas ferramentas assumem este nome de banco de dados como sendo o padrão, evitando a necessidade de digitá-lo. Para criar este banco de dados digite simplesmente

```
$ createdb
```

Se você não deseja mais utilizar o seu banco de dados você pode removê-lo. Por exemplo, se você for o dono (criador) do banco de dados `meu_bd`, você poderá removê-lo utilizando o seguinte comando:

```
$ dropdb meu_bd
```

Para este comando o nome da conta não é utilizado como o nome padrão do banco de dados, o nome sempre deve ser especificado. Esta ação remove fisicamente todos os arquivos associados ao banco de dados e não pode ser desfeita, portanto esta operação somente deve ser feita após um longo período de reflexão.

1.4. Acessando um banco de dados

Após o banco de dados ter sido criado, este pode ser acessado pela:

- Execução do programa de terminal interativo do PostgreSQL chamado *psql*, que permite a entrada, edição e execução de comandos SQL interativos.
- Utilização de uma ferramenta cliente gráfica existente como o PgAccess, ou de um pacote de automação de escritórios com suporte a ODBC para criar e manusear bancos de dados. Estas possibilidades não estão cobertas neste tutorial.
- Criação de aplicativos personalizados, usando uma das várias ligações com linguagens disponíveis. Estas possibilidades são discutidas com mais detalhe no *Guia do Programador do PostgreSQL*.

Você provavelmente deseja iniciar o `psql` para executar os exemplos deste tutorial. O `psql` pode ser ativado para usar o banco de dados `meu_bd` digitando o seguinte comando:

```
$ psql meu_bd
```

Se o nome do banco de dados for omitido, então o padrão fica sendo o nome da conta de seu usuário. Você já viu este esquema na seção anterior.

No `psql` você vai ser saudado com a seguinte mensagem:

```
Welcome to psql 7.3.2, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
```

1. Uma explicação de porque isto funciona: Os nomes dos usuários do PostgreSQL são distintos dos nomes das contas do sistema operacional. Ao se conectar com um banco de dados pode ser escolhido o nome do usuário do PostgreSQL que deseja se conectar como; Se isto não for feito isto, o padrão é utilizar o mesmo nome da conta atual do sistema operacional. Como isto ocorre, sempre existirá uma conta de usuário do PostgreSQL que possui o mesmo nome do usuário do sistema operacional que inicializou o servidor e acontece, também, que este usuário sempre tem permissão para criar bancos de dados. Em vez de se logar como o usuário pode ser especificada a opção `-U` em todos os aplicativos para escolher o nome do usuário do PostgreSQL que se deseja conectar como.

```
        \h for help with SQL commands
        \? for help on internal slash commands
        \g or terminate with semicolon to execute query
        \q to quit

meu_bd=>
```

A última linha também pode ser

```
meu_bd=#
```

Isto significa que você é um super-usuário do banco de dados, o que acontece geralmente quando você instala o PostgreSQL por si mesmo. Sendo um super-usuário significa que você não está sujeito a controles de acesso. Para as finalidades deste tutorial isto não tem importância.

Se você teve problemas ao inicializar o `psql`, então retorne à seção anterior. Os diagnósticos do `psql` e do `createdb` são similares, e se um funcionou o outro deve funcionar também.

A última linha exibida pelo `psql` é o `prompt`, indicando que o `psql` está aguardando você, e que você pode digitar comandos SQL dentro do espaço de trabalho mantido pelo `psql`. Tente usar estes comando:

```
meu_bd=> SELECT version();
                            version
----------------------------------------------------------------
 PostgreSQL 7.3devel on i586-pc-linux-gnu, compiled by GCC 2.96
(1 row)

meu_bd=> SELECT current_date;
    date
------------
 2002-08-31
(1 row)

meu_bd=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

O programa `psql` possui vários comandos internos que não são comandos SQL. Eles começam pelo caractere de contrabarra, “\”. Alguns destes comandos são mostrados na mensagem de boas vindas. Por exemplo, pode ser obtida ajuda na sintaxe de vários comandos SQL do PostgreSQL digitando-se:

```
meu_bd=> \h
```

Para sair do `psql` digite

```
meu_bd=> \q
```

e o `psql` vai terminar e retornar para a linha de comando do sistema operacional (para conhecer outros comandos internos digite `\?` no `prompt` do `psql`). Todas as funcionalidades do `psql` estão documentadas no *Manual de Referência do PostgreSQL*. Se o PostgreSQL estiver instalado corretamente, você também poderá digitar `man psql` na linha de comando do sistema operacional para ver a documentação. Neste tutorial nós não vamos utilizar estas funcionalidades explicitamente, mas você poderá usá-las por si próprio quando julgar adequado.

Capítulo 2. A linguagem SQL

2.1. Introdução

Este capítulo contém uma revisão sobre como utilizar a linguagem SQL para realizar operações simples. Este tutorial tem apenas o propósito de fazer uma introdução e, de forma alguma, é um tutorial completo sobre a linguagem SQL. Muitos livros foram escritos sobre o SQL, incluindo o *Understanding the New SQL* e *A Guide to the SQL Standard*. Você deve ficar ciente de que alguma funcionalidades da linguagem no PostgreSQL são extensões ao padrão.

Nos exemplos a seguir é assumido que você tenha criado o banco de dados chamado `meu_bd`, conforme descrito no capítulo anterior, e que tenha ativado o psql.

Os exemplos presentes neste manual também podem ser encontrados no fonte da distribuição do PostgreSQL no diretório `src/tutorial/`. Consulte o arquivo `README` neste diretório para saber como usá-los. Para iniciar o tutorial (em inglês) faça o seguinte:

```
$ cd ..../src/tutorial
$ psql -s mydb
...

mydb=> \i basics.sql
```

O comando `\i` lê os comandos no arquivo especificado. A opção `-s` ativa o modo passo a passo, que faz uma pausa antes de enviar cada comando para o servidor. Os comandos utilizados nesta seção estão no arquivo `basics.sql`.

2.2. Conceitos

O PostgreSQL é um *sistema de gerenciamento de banco de dados relacional* (SGBDR). Isto significa que é um sistema para gerenciar dados armazenados em *relações*. Uma relação é essencialmente um termo matemático para *tabela*. A noção de armazenar dados em tabelas é um lugar tão comum hoje em dia que pode parecer totalmente óbvio, mas existem várias outras maneiras de se organizar bancos de dados. Arquivos e diretórios em sistemas operacionais tipo Unix são um exemplo de banco de dados hierárquico. Um desenvolvimento mais moderno são os bancos de dados orientados a objeto.

Cada tabela é uma coleção nomeada de *linhas*. Cada linha de uma determinada tabela possui o mesmo conjunto de *colunas* nomeadas, e cada coluna é de um tipo de dado específico. Enquanto as colunas possuem uma ordem fixa em cada linha, é importante lembrar que o SQL não garante a ordem das linhas dentro de uma tabela (embora as linhas possam ser explicitamente ordenadas para a exibição).

As tabelas são agrupadas em bancos de dados, e uma coleção de bancos de dados gerenciada por uma única instância do servidor PostgreSQL constitui um *agrupamento* de bancos de dados.

2.3. Criando uma nova tabela

Uma nova tabela pode ser criada especificando-se o seu nome juntamente com os nomes das colunas e seus tipos:

```
CREATE TABLE clima (
    cidade          varchar(80),
    temp_min        int,           -- temperatura mínima
```

```
    temp_max        int,           -- temperatura máxima
    prcp            real,          -- precipitação
    data            date
);
```

O comando pode ser digitado no `psql` com as quebras de linha. O `psql` vai reconhecer que o comando ainda não terminou até encontrar o ponto-e-vírgula.

Espaços em branco (ou seja, espaços, tabulações e novas linhas) podem ser utilizadas livremente nos comandos SQL. Isto significa que o comando pode ser digitado com um alinhamento diferente do mostrado acima, ou mesmo todo em uma única linha. Dois hífens ("`--`") introduzem um comentário. Tudo que segui-los é ignorado até o final da linha. O SQL não diferencia letras maiúsculas e minúsculas nas palavras e nos identificadores, a não ser quando os identificadores estão entre aspas (`"`) para preservar letras maiúsculas e minúsculas (o que não foi feito acima).

O `varchar(80)` especifica um tipo de dado que pode armazenar cadeias de caracteres arbitrárias com um comprimento de até 80 caracteres; O `int` é o tipo inteiro normal; O `real` é um tipo para armazenar números de ponto flutuante de precisão simples; `date` é para armazenar data e hora (a coluna do tipo `date` poderia se chamar `date`, o que tanto pode ser conveniente quanto pode causar confusão).

O PostgreSQL suporta os tipos SQL usuais `int`, `smallint`, `real`, `double precision`, `char(`*`N`*`)`, `varchar(`*`N`*`)`, `date`, `time`, `timestamp` e `interval`, assim bem como outros tipos de utilidade geral e um abrangente conjunto de tipos geométricos. O PostgreSQL pode ser personalizado com um número arbitrário de tipos definidos pelo usuário. Como conseqüência os nomes dos tipos não são sintaticamente palavras chaves, exceto onde for requerido para suportar casos especiais no padrão SQL.

No segundo exemplo são armazenadas as cidades e suas localizações geográficas:

```
CREATE TABLE cidades (
    nome            varchar(80),
    localizacao     point
);
```

O tipo `point` é um exemplo de tipo específico do PostgreSQL.

Finalmente, deve ser mencionado que se a tabela não for mais necessária, ou se deseja recriá-la de uma forma diferente, esta pode ser removida através do seguinte comando:

```
DROP TABLE nome_da_tabela;
```

2.4. Povoando uma tabela com linhas

O comando `INSERT` é utilizado para povoar uma tabela com linhas:

```
INSERT INTO clima VALUES ('São Francisco', 46, 50, 0.25, '1994-11-27');
```

Repare que todos os tipos de dados usam formatos de entrada de dados bastante óbvios. As constantes, que não são apenas valores numéricos, geralmente devem estar entre apóstrofos (`'`), como no exemplo acima. O tipo `date` na verdade é muito flexível com relação aos dados que aceita, mas para este tutorial será utilizado o formato mostrado acima porque não possui ambigüidade.

O tipo `point` requer um par de coordenadas como entrada, como mostrado abaixo:

```
INSERT INTO cidades  VALUES ('São Francisco', '(-194.0, 53.0)');
```

A sintaxe usada anteriormente requer que seja lembrada a ordem das colunas. Uma sintaxe alternativa permite relacionar as colunas explicitamente:

```
INSERT INTO clima (cidade, temp_min, temp_max, prcp, data)
    VALUES ('São Francisco', 43, 57, 0.0, '1994-11-29');
```

As colunas podem ser listadas em uma ordem diferente se for desejado, ou até mesmo omitir algumas colunas. Por exemplo, se a precipitação for desconhecida:

```
INSERT INTO clima (data, cidade, temp_max, temp_min)
    VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Muitos desenvolvedores consideram relacionar explicitamente as colunas um estilo melhor do que confiar na ordem implícita.

Por favor, entre todos os comando mostrados acima para ter alguns dados com que trabalhar nas próximas seções.

Também pode ser utilizado o comando `COPY` para carregar uma grande quantidade de dados a partir de arquivos de texto. Geralmente isto é mais rápido porque o comando `COPY` é otimizado para esta finalidade, embora possua menos flexibilidade do que o comando `INSERT`. Um exemplo poderia ser:

```
COPY clima FROM '/home/user/clima.txt';
```

onde o arquivo de origem deve poder ser acessado pelo servidor e não pelo o cliente, uma vez que o servidor lê o arquivo diretamente. Pode-se obter mais informações sobre o comando `COPY` no *Manual de Referência do PostgreSQL*.

2.5. Consultando uma tabela

Para obter os dados de uma tabela, a tabela deve ser *consultada*. O comando `SELECT` do SQL é utilizado para esta função. O comando é dividido em lista de seleção (a parte que relaciona as colunas a serem retornadas), lista de tabelas (a parte que relaciona as tabelas de onde os dados vão ser extraídos), e uma qualificação opcional (a parte em que são especificadas as restrições). Por exemplo, para ver todas as linhas da tabela `clima` digite:

```
SELECT * FROM clima;
```

(aqui o `*` significa "todas as colunas") e a saída deve ser:

```
     cidade      | temp_min | temp_max | prcp |    data
-----------------+----------+----------+------+------------
 São Francisco   |       46 |       50 | 0.25 | 1994-11-27
 São Francisco   |       43 |       57 |    0 | 1994-11-29
 Hayward         |       37 |       54 |      | 1994-11-29
(3 rows)
```

Pode ser especificada qualquer expressão arbitrária na lista de seleção. Por exemplo, pode-se escrever:

```
SELECT cidade, (temp_max+temp_min)/2 AS temp_media, data FROM clima;
```

O que deve produzir:

```
     cidade      | temp_media |    data
-----------------+------------+------------
```

```
 São Francisco   |         48 | 1994-11-27
 São Francisco   |         50 | 1994-11-29
 Hayward         |         45 | 1994-11-29
(3 rows)
```

Perceba que a cláusula AS foi utilizada para mudar o nome da coluna de saída (é opcional).

Operadores booleanos arbitrários (AND, OR e NOT) são permitidos na qualificação da consulta. Por exemplo, o comando abaixo obtém o clima de São Francisco nos dias de chuva:

```
SELECT * FROM clima
    WHERE cidade = 'São Francisco'
    AND prcp > 0.0;
```

Resultado:

```
     cidade      | temp_min | temp_max | prcp |    data
-----------------+----------+----------+------+------------
 São Francisco   |       46 |       50 | 0.25 | 1994-11-27
(1 row)
```

Como nota final, pode-se desejar que os resultados de uma seleção retornem em uma determinada ordem, ou com as linhas duplicadas removidas:

```
SELECT DISTINCT cidade
    FROM clima
    ORDER BY cidade;
```

```
     cidade
---------------
 Hayward
 São Francisco
(2 rows)
```

As cláusulas DISTINCT e ORDER BY podem ser usadas separadamente, é claro.

2.6. Junções entre tabelas

Até agora nossas consultas somente acessaram uma tabela de cada vez. As consultas podem acessar várias tabelas de uma vez, ou acessar a mesma tabela de uma maneira que várias linhas da tabela são processadas ao mesmo tempo. Uma consulta que acessa várias linhas da mesma tabela ou de tabelas diferentes de uma vez é chamada de consulta de *junção*. Como exemplo, digamos que se deseja listar todos os registros de clima junto com a localização da cidade associada. Para fazer isto, necessitamos comparar a coluna cidade de cada linha da tabela clima com a coluna nome de todas as linhas da tabela cidades, e selecionar os pares de linha onde estes valores são correspondentes.

> **Nota:** Este é apenas um modelo conceitual, a junção real pode ser realizada de uma maneira mais eficiente, mas isto não é visível para o usuário.

Esta operação pode ser efetuada através da seguinte consulta:

```
SELECT *
    FROM clima, cidades
    WHERE cidade = nome;
```

```
     cidade      | temp_min | temp_max | prcp |    data    |     nome      | localizacao
-----------------+----------+----------+------+------------+---------------+-------------
 São Francisco   |       46 |       50 | 0.25 | 1994-11-27 | São Francisco | (-194,53)
 São Francisco   |       43 |       57 |    0 | 1994-11-29 | São Francisco | (-194,53)
(2 rows)
```

Observe duas coisas no resultado produzido:

- Não existe nenhuma linha para a cidade Hayward. Isto é porque não existe entrada correspondente na tabela `cidades` para Hayward, portanto a junção ignora as linhas da tabela clima sem correspondência. Nós veremos em breve como isto pode ser mudado.
- Existem duas colunas contendo o nome da cidade. Isto está correto porque a relação de colunas das tabelas `clima` e `cidades` estão concatenadas. Na prática isto não é o que se deseja e, portanto, é preferível escrever a lista das colunas de saída explicitamente em vez de utilizar o `*`:

```
SELECT cidade, temp_min, temp_max, prcp, data, localizacao
    FROM clima, cidades
    WHERE cidade = nome;
```

Exercício: Tente descobrir a semântica desta consulta quando a cláusula `WHERE` é omitida.

Uma vez que todas as colunas possuem nomes diferentes, o analisador encontra automaticamente a tabela que a coluna pertence, mas é um bom estilo qualificar completamente os nomes das colunas nas consultas de junção:

```
SELECT clima.cidade, clima.temp_min, clima.temp_max,
       clima.prcp, clima.data, cidades.localizacao
    FROM clima, cidades
    WHERE cidades.nome = clima.cidade;
```

As consultas de junção do tipo visto até agora também poderiam ser escritas da seguinte forma alternativa:

```
SELECT *
    FROM clima INNER JOIN cidades ON (clima.cidade = cidades.nome);
```

A utilização desta sintaxe não é tão comum quanto das demais acima, mas é mostrada aqui para que sejam entendidos os próximos tópicos.

Agora nós vamos descobrir como se faz para obter os registros de Hayward. O que nós queremos que seja feito é varrer a tabela `clima`, e para cada linha encontrar a linha correspondente em `cidades`. Se nenhuma linha for encontrada, nós queremos que algum “valor vazio” seja colocado nas colunas da tabela `cidades`. Este tipo de consulta é chamado de *junção externa* (`outer join`). As consultas vistas até agora são junções internas (`inner join`). O comando então fica assim:

```
SELECT *
    FROM clima LEFT OUTER JOIN cidades ON (clima.cidade = cidades.nome);

     cidade      | temp_min | temp_max | prcp |    data    |     nome      | localizacao
-----------------+----------+----------+------+------------+---------------+------------
 Hayward         |       37 |       54 |      | 1994-11-29 |               |
 São Francisco   |       46 |       50 | 0.25 | 1994-11-27 | São Francisco | (-194,53)
 São Francisco   |       43 |       57 |    0 | 1994-11-29 | São Francisco | (-194,53)
(3 rows)
```

Esta consulta é chamada de *junção externa esquerda* (`left outer join`) porque a tabela mencionada à esquerda do operador de junção terá cada uma de suas linhas aparecendo na saída ao menos uma vez, enquanto que a tabela à direita vai ter somente as linhas que correspondem a alguma linha da tabela à esquerda aparecendo. Ao listar uma linha da tabela à esquerda, para a qual não existe nenhuma linha correspondente na tabela à direita, valores vazios (`null`) são colocados nas colunas da tabela à direita.

Exercício: Existe também a junção externa direita (`right outer join`) e a junção externa completa (`full outer join`). Tente descobrir o que fazem.

Também é possível fazer a junção da tabela com si mesma. Isto é chamado de *autojunção* (`self join`). Como exemplo, suponha que nós desejamos descobrir todos os registro de clima que estão no intervalo de temperatura de outros registros de clima. Portanto, nós precisamos comparar as colunas `temp_min` e `temp_max` de cada linha de `clima` com as colunas `temp_min` e `temp_max` de todas as outras linhas da tabela `clima`. Nós podemos fazer isto utilizando a seguinte consulta:

```
SELECT C1.cidade, C1.temp_min AS menor, C1.temp_max AS maior,
    C2.cidade, C2.temp_min AS menor, C2.temp_max AS maior
    FROM clima C1, clima C2
    WHERE C1.temp_min < C2.temp_min
    AND C1.temp_max > C2.temp_max;

     cidade      | menor | maior |     cidade    | menor | maior
-----------------+-------+-------+---------------+-------+-------
 São Francisco   |    43 |    57 | São Francisco |    46 |    50
 Hayward         |    37 |    54 | São Francisco |    46 |    50
(2 rows)
```

A tabela clima teve o seu nome mudado para `C1` e `C2` para que se possa distinguir o lado esquerdo e o direito da junção. Estes “aliases” também podem ser utilizados em outras consultas para reduzir a digitação como, por exemplo:

```
SELECT *
    FROM clima w, cidades c
    WHERE w.cidade = c.nome;
```

Será encontrada esta forma de abreviar com bastante freqüência.

2.7. Funções de agregação

Como a maioria dos produtos de banco de dados relacional, o PostgreSQL suporta funções de agregação. Uma função de agregação calcula um único resultado para várias linhas de entrada. Por exemplo, existem funções de agregação para contar (`count`), somar (`sum`), calcular a média (`avg`), o valor máximo (`max`) e o valor mínimo (`min`) para um conjunto de linhas.

Como exemplo podemos obter a maior temperatura mínima ocorrida em qualquer lugar com

```
SELECT max(temp_min) FROM clima;

 max
-----
  46
(1 row)
```

Se desejarmos saber em que cidade (ou cidades) esta leitura ocorreu, podemos tentar

```
SELECT cidade FROM clima WHERE temp_min = max(temp_min);     ERRADO
```

mas não vai funcionar porque a função de agregação `max` não pode ser usada na cláusula `WHERE` (Esta restrição existe porque a cláusula `WHERE` determina as linhas que passarão para o estágio de agregação; portanto tem que ser avaliada antes das funções de agregação serem computadas). Entretanto, como é geralmente o caso, a consulta pode ser reformulada para obter o resultado pretendido, o que será feito aqui através de uma *subconsulta*:

```
SELECT cidade FROM clima
    WHERE temp_min = (SELECT max(temp_min) FROM clima);

     cidade
---------------
 São Francisco
(1 row)
```

Isto está correto porque a subconsulta é uma ação independente, que calcula a sua própria agregação separadamente do que está acontecendo na consulta externa.

As agregações também são muito úteis quando combinadas com a cláusula `GROUP BY`. Por exemplo, pode ser obtida a maior temperatura mínima observada em cada cidade com

```
SELECT cidade, max(temp_min)
    FROM clima
    GROUP BY cidade;

     cidade      | max
-----------------+-----
 Hayward         |  37
 São Francisco   |  46
(2 rows)
```

o que produz uma linha de saída para cada cidade. Cada resultado da agregação é calculado sobre as linhas da tabela que correspondem a uma cidade. Estas linhas agrupadas podem ser filtradas utilizando a cláusula `HAVING`:

```
SELECT cidade, max(temp_min)
    FROM clima
    GROUP BY cidade
    HAVING max(temp_min) < 40;

  cidade   | max
-----------+-----
 Hayward   |  37
(1 row)
```

que nos fornece o mesmo resultado apenas para as cidades que possuam todos os valores de `temp_min` abaixo de 40. Finalmente, se nós desejamos somente as cidade que começam com a letra “`S`”, podemos escrever

```
SELECT cidade, max(temp_min)
    FROM clima
    WHERE cidade LIKE ’S%’❶
    GROUP BY cidade
    HAVING max(temp_min) < 40;
```

❶ O operador `LIKE` faz correspondência de padrão e é explicado no *Guia do Usuário do PostgreSQL*.

É importante compreender a interação entre as agregações e as cláusulas WHERE e HAVING do SQL. A diferença fundamental entre WHERE e HAVING é esta: o WHERE seleciona as linhas de entrada antes dos grupos e agregações serem computados (portanto, controla quais linhas irão para o computo da agregação), enquanto que o HAVING seleciona grupos de linhas após os grupos e agregações serem computados. Portanto, a cláusula WHERE não pode conter funções de agregação; não faz sentido tentar utilizar uma agregação para determinar quais linhas serão a entrada da agregação. Por outro lado, a cláusula HAVING sempre possui função de agregação (Falando estritamente, é permitido escrever uma cláusula HAVING que não possua agregação, mas é desperdício: A mesma condição poderia ser utilizada de forma mais eficiente no estágio do WHERE).

Observe que a restrição do nome da cidade poderia ser colocada na cláusula WHERE, uma vez que não necessita de nenhuma agregação. Isto é mais eficiente do que colocar a restrição na cláusula HAVING, porque evita que sejam feitos os procedimentos de agrupamento e agregação para todas as linhas que não atendem a cláusula WHERE.

2.8. Atualizações

As linhas existentes podem ser atualizadas usando-se o comando UPDATE. Suponha que seja descoberto que as leituras de temperatura estejam todas 2 graus acima após 28 de Novembro de 1994. Estes dados podem ser corrigidos da seguinte maneira:

```
UPDATE clima
    SET temp_max = temp_max - 2,  temp_min = temp_min - 2
    WHERE data > ’1994-11-28’;
```

Agora vejamos o novo estado dos dados:

```
SELECT * FROM clima;

     cidade      | temp_min | temp_max | prcp |    data
-----------------+----------+----------+------+------------
 São Francisco   |       46 |       50 | 0.25 | 1994-11-27
 São Francisco   |       41 |       55 |    0 | 1994-11-29
 Hayward         |       35 |       52 |      | 1994-11-29
(3 rows)
```

2.9. Exclusões

Suponha que nós não estamos mais interessados nos dados do clima em Hayward. Então precisamos excluir estas linhas da tabela. As exclusões são realizadas através do comando DELETE:

```
DELETE FROM clima WHERE cidade = ’Hayward’;
```

Todos os registros de clima pertencentes a Hayward são removidos.

```
SELECT * FROM clima;

    cidade     | temp_min | temp_max | prcp |    data
---------------+----------+----------+------+------------
 São Francisco |       46 |       50 | 0.25 | 1994-11-27
 São Francisco |       41 |       55 |    0 | 1994-11-29
```

```
(2 rows)
```

Deve-se tomar cuidado com a forma

```
DELETE FROM nome_da_tabela;
```

Sem uma qualificação, o comando `DELETE` remove *todas* as linhas de uma dada tabela, deixando-a vazia. O sistema não vai solicitar uma confirmação antes de realizar esta operação!

Capítulo 3. Funcionalidades avançadas

3.1. Introdução

Nos capítulos anteriores foram descritas funcionalidade básicas da utilização do SQL para armazenamento e acesso aos dados no PostgreSQL. Agora serão discutidas funcionalidades mais avançadas do SQL que simplificam a gerência e evitam a perda e a corrupção dos dados. Ao final serão vistas algumas extensões do PostgreSQL.

Este capítulo em certas ocasiões faz referência aos exemplos encontrados no Capítulo 2 para modificá-los ou melhorá-los, portanto é aconselhável que você já tenha lido este capítulo. Alguns exemplos deste capítulo também podem ser encontrados no arquivo `advanced.sql` no diretório do tutorial. Este arquivo também contém alguns dados de exemplo a serem carregados, que não serão repetidos aqui (consulte a Seção 2.1 para saber como usar este arquivo).

3.2. Visões

Reveja as consultas na Seção 2.6. Suponha que a consulta combinando os registros de clima e de localização das cidades seja de particular interesse para o seu aplicativo, mas que você não deseja digitar esta consulta toda vez que necessitar dela. Pode-se então criar uma *visão* baseada na consulta, a qual dá um nome à consulta, através do qual pode ser feita referência como se fosse uma tabela comum.

```
CREATE VIEW minha_visao AS
    SELECT cidade, temp_min, temp_max, prcp, data, localizacao
        FROM clima, cidades
        WHERE cidade = nome;

SELECT * FROM minha_visao;
```

Fazer livre uso de visões é um aspecto chave de um bom projeto de banco de dados SQL. As visões permitem encapsular os detalhes da estrutura das tabelas, que podem mudar na medida em que os aplicativos evoluem, atrás de interfaces consistentes.

As visões podem ser utilizadas em praticamente todos os lugares em que uma tabela pode ser utilizada. Construir visões baseadas em visões não é raro.

3.3. Chaves estrangeiras

Reveja as tabelas `clima` e `cidades` no Capítulo 2. Considere o seguinte problema: Desejamos ter certeza que não serão inseridas linhas na tabela `clima` sem que haja uma entrada correspondente na tabela `cidades`. Isto é chamado de manter a *integridade referencial* dos dados. Em sistemas de banco de dados muito simples isto poderia ser implementado (caso fosse) primeiro olhando-se a tabela `cidades` para ver se um registro correspondente existe, e depois inserindo ou rejeitando o novo registro de `clima`. Esta abordagem possui vários problemas e é muito inconveniente, portanto o PostgreSQL pode realizar esta operação por você.

As novas declarações para as tabelas ficariam assim:

```
CREATE TABLE cidades (
  cidade      varchar(80) primary key,
```

```
 localizacao point
);

CREATE TABLE clima (
 cidade     varchar(80) references cidades,
 temp_min   int,
 temp_max   int,
 prcp       real,
 data       date
);
```

Agora ao se tentar inserir um registro inválido:

```
INSERT INTO clima VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');

ERROR: <unnamed> referential integrity violation - key referenced from clima not found in ci
```

O comportamento das chaves estrangeiras pode receber um ajuste fino na aplicação. Não iremos além deste exemplo simples neste tutorial, mas consulte o *Guia do Usuário do PostgreSQL* para obter mais informações. Usar corretamente as chaves estrangeiras com certeza vai melhorar a qualidade dos seus aplicativos de banco de dados, portanto encorajamos muito que este tópico seja aprendido.

3.4. Transações

Transação é um conceito fundamental de todo o sistema de banco de dados. O ponto essencial de uma transação é que esta engloba vários passos em uma única operação de tudo ou nada. Os estados dos passos intermediários não são visíveis para as outras transações concorrentes e, se alguma falha ocorrer que impeça a transação de chegar até o fim, então nenhum dos passos intermediários irá afetar o banco de dados de nenhuma forma.

Por exemplo, considere um banco de dados de uma instituição financeira que contém o saldo da conta corrente de vários clientes, assim como o saldo total dos depósitos de cada agência. Suponha que se deseje transferir $100.00 da conta da Alice para a conta do Bob. Simplificando barbaramente, os comandos SQL para esta operação seriam:

```
UPDATE conta_corrente SET saldo = saldo - 100.00
    WHERE nome = 'Alice';
UPDATE filiais SET saldo = saldo - 100.00
    WHERE nome = (SELECT nome_filial FROM conta_corrente WHERE nome = 'Alice');
UPDATE conta_corrente SET saldo = saldo + 100.00
    WHERE nome = 'Bob';
UPDATE filiais SET saldo = saldo + 100.00
    WHERE nome = (SELECT nome_filial FROM conta_corrente WHERE nome = 'Bob');
```

Os detalhes destes comandos não são importantes aqui; o importante é que existem várias atualizações separadas envolvidas para realizar esta operação bem simples. A contabilidade do banco quer ter certeza que todas as atualizações são realizadas, ou que nenhuma delas acontece. Não é interessante uma falha no sistema fazer com que Bob receba $100.00 que não foi debitado da Alice. A Alice também não vai continuar sendo uma cliente satisfeita se o dinheiro for debitado da conta dela e não for creditado na de Bob. É necessário garantir que se algo der errado no meio da operação, nenhum dos passos executados até então surtam efeito. Agrupando-se estas atualizações em uma *transação* nos dá esta garantia. Uma transação é dita como sendo *atômica*: do ponto de vista das outras transações, ou ela acontece plenamente ou nada acontece.

Desejamos também a garantia de uma vez que a transação esteja completa e aceita pelo sistema de banco de dados, que fique permanentemente gravada e não seja perdida mesmo no caso de uma pane acontecer logo em seguida. Por exemplo, se estiver sendo registrado um saque em dinheiro pelo Bob, não é desejado de forma alguma que o débito em sua conta corrente desapareça devido a uma pane ocorrida logo depois dele sair do banco. Um banco de dados transacional garante que todas as atualizações feitas por uma transação estão registradas em um meio de armazenamento permanente (ou seja, em disco) antes da transação ser considerada completa.

Uma outra propriedade importante dos bancos de dados transacionais está muito ligada à noção de atualizações atômicas: quando várias transações estão executando ao mesmo tempo, cada uma delas não deve enxergar as mudanças intermediárias efetuadas pelas outras. Por exemplo, se uma transação está ocupada totalizando o saldo de todas as agências, não pode ser visto o débito efetuado na agência da Alice mas ainda não creditado na agência do Bob, nem o contrário. Portanto, as transações não devem ser tudo ou nada apenas em termos de seu efeito permanente no banco de dados, mas também em termos de visibilidade durante o processamento. As atualizações feitas por uma transação não podem ser vistas pelas outras transações enquanto esta não terminar, quando todas as atualizações se tornam visíveis ao mesmo tempo.

No PostgreSQL uma transação é definida cercando-se os comandos SQL da transação com os comandos `BEGIN` e `COMMIT`. Sendo assim, a nossa transação bancária ficaria:

```
BEGIN;
UPDATE conta_corrente SET saldo = saldo - 100.00
    WHERE nome = 'Alice';
-- etc etc
COMMIT;
```

Se no meio da transação for decidido que esta não deve ser concluída (talvez porque foi visto que o saldo da Alice se tornou negativo), pode ser executado o comando `ROLLBACK` em vez do `COMMIT`, fazendo com que todas as atualizações sejam canceladas.

O PostgreSQL na verdade trata todo comando SQL como sendo executado dentro de uma transação. Se não for utilizado o comando `BEGIN`, então cada comando possui individualmente um `BEGIN` e um (se der tudo certo) `COMMIT` em torno dele. Um grupo de comandos envolvidos por um `BEGIN` e um `COMMIT` é algumas vezes chamado de *bloco de transação*.

> **Nota:** Algumas bibliotecas cliente emitem um comando `BEGIN` e um comando `COMMIT` automaticamente, fazendo com que se obtenha o efeito de um bloco de transação sem que seja perguntado. Verifique a documentação da interface sendo utilizada.

3.5. Herança

Herança é um conceito de banco de dados orientado a objeto que abre novas possibilidades interessantes ao projeto de bancos de dados.

Criemos duas tabelas: uma tabela `cidades` e outra tabela `capitais`. Naturalmente as capitais também são cidades e, portanto, deve existir alguma maneira de se mostrar implicitamente as capitais quando todas as cidades são mostradas. Sendo bastante perspicaz pode-se criar um esquema como este:

```
CREATE TABLE capitais (
  nome       text,
  populacao  real,
  altitude   int,    -- (em pés)
```

```
  estado     char(2)
);

CREATE TABLE interior (
  nome       text,
  populacao  real,
  altitude   int     -- (em pés)
);

CREATE VIEW cidades AS
  SELECT nome, populacao, altitude FROM capitais
    UNION
  SELECT nome, populacao, altitude FROM interior;
```

Este esquema funciona bem para as consultas, mas não é bom quando é necessário atualizar várias linhas, entre outras coisas.

Uma solução melhor é esta:

```
CREATE TABLE cidades (
  nome       text,
  populacao  real,
  altitude   int     -- (em pés)
);

CREATE TABLE capitais (
  estado      char(2)
) INHERITS (cidades);
```

Neste caso, uma linha de `capitais` *herda* todas as colunas (`nome`, `populacao` e `altitude`) de seu *ancestral* `cidades`. O tipo da coluna `nome` é `text`, um tipo nativo do PostgreSQL para cadeias de caracteres de tamanho variável. As capitais dos estados têm uma coluna a mais chamada estado, que guarda a sigla do estado. No PostgreSQL uma tabela pode herdar de nenhuma ou de várias tabelas.

Por exemplo, a consulta abaixo lista os nomes de todas as cidades, incluindo as capitais dos estados, que estão localizadas a uma altitude superior a 500 pés:

```
SELECT name, altitude
  FROM cities
  WHERE altitude > 500;
```

o que retorna:

```
   nome    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
(3 rows)
```

Por outro lado, a consulta abaixo lista todas as cidades que não são capitais e estão situadas a uma altitude superior a 500 pés:

```
SELECT nome, altitude
```

```
    FROM ONLY cidades
    WHERE altitude > 500;

   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 rows)
```

Nesta consulta a palavra `ONLY` antes de `cidades` indica que a consulta deve ser efetuada somente na tabela `cidades`, e não nas tabelas hierarquicamente abaixo de `cidades` em termos de herança. Muitos comandos discutidos até agora -- `SELECT`, `UPDATE` e `DELETE` -- permitem usar a notação do `ONLY`.

3.6. Conclusão

O PostgreSQL possui muitas funcionalidades não descritas neste tutorial introdutório, o qual está orientado para os usuários novatos em SQL. Estas funcionalidades estão discutidas com mais detalhes no *Guia do Usuário do PostgreSQL* e no *Guia do Programador do PostgreSQL*.

Se você achar que necessita de mais material introdutório, por favor visite o sítio do PostgreSQL na Web[1] para obter mais informações.

1. http://www.postgresql.org

Bibliografia

Referências e artigos selecionados para o SQL e para o PostgreSQL.

Alguns relatórios oficiais e técnicos da equipe original de desenvolvimento do POSTGRES estão disponíveis no sítio na Web do Departamento de Ciência da Computação da Universidade da Califórnia em Berkeley.[1]

Livros de Referência sobre o SQL

Judith Bowman, Sandra Emerson, e Marcy Darnovsky, *The Practical SQL Handbook: Using Structured Query Language*, Third Edition, Addison-Wesley, ISBN 0-201-44787-8, 1996.

C. J. Date e Hugh Darwen, *A Guide to the SQL Standard: A user's guide to the standard database language SQL*, Fourth Edition, Addison-Wesley, ISBN 0-201-96426-0, 1997.

C. J. Date, *An Introduction to Database Systems*, Volume 1, Sixth Edition, Addison-Wesley, 1994.

Ramez Elmasri e Shamkant Navathe, *Fundamentals of Database Systems*, 3rd Edition, Addison-Wesley, ISBN 0-805-31755-4, August 1999.

Jim Melton e Alan R. Simon, *Understanding the New SQL: A complete guide*, Morgan Kaufmann, ISBN 1-55860-245-3, 1993.

Jeffrey D. Ullman, *Principles of Database and Knowledge: Base Systems*, Volume 1, Computer Science Press, 1988.

Documentação específica do PostgreSQL

Stefan Simkovics, *Enhancement of the ANSI SQL Implementation of PostgreSQL*, Department of Information Systems, Vienna University of Technology, November 29, 1998.

Discusses SQL history and syntax, and describes the addition of `INTERSECT` and `EXCEPT` constructs into PostgreSQL. Prepared as a Master's Thesis with the support of O. Univ. Prof. Dr. Georg Gottlob and Univ. Ass. Mag. Katrin Seyr at Vienna University of Technology.

A. Yu e J. Chen, The POSTGRES Group, *The Postgres95 User Manual*, University of California, Sept. 5, 1995.

Zelaine Fong, *The design and implementation of the POSTGRES query optimizer*[2], University of California, Berkeley, Computer Science Department.

1. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/
2. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/UCB-MS-zfong.pdf

Conferências e Artigos

Nels Olson, *Partial indexing in POSTGRES: research project*, University of California, UCB Engin T7.49.1993 O676, 1993.

L. Ong e J. Goh, “A Unified Framework for Version Modeling Using Production Rules in a Database System”, *ERL Technical Memorandum M90/33*, University of California, April, 1990.

L. Rowe e M. Stonebraker, “The POSTGRES data model[3]”, Proc. VLDB Conference, Sept. 1987.

P. Seshadri e A. Swami, “Generalized Partial Indexes[4] ”, Proc. Eleventh International Conference on Data Engineering, 6-10 March 1995, IEEE Computer Society Press, Cat. No.95CH35724, 1995, p. 420-7.

M. Stonebraker e L. Rowe, “The design of POSTGRES[5]”, Proc. ACM-SIGMOD Conference on Management of Data, May 1986.

M. Stonebraker, E. Hanson, e C. H. Hong, “The design of the POSTGRES rules system”, Proc. IEEE Conference on Data Engineering, Feb. 1987.

M. Stonebraker, “The design of the POSTGRES storage system[6]”, Proc. VLDB Conference, Sept. 1987.

M. Stonebraker, M. Hearst, e S. Potamianos, “A commentary on the POSTGRES rules system[7]”, *SIGMOD Record 18(3)*, Sept. 1989.

M. Stonebraker, “The case for partial indexes[8]”, *SIGMOD Record 18(4)*, Dec. 1989, p. 4-11.

M. Stonebraker, L. A. Rowe, e M. Hirohama, “The implementation of POSTGRES[9]”, *Transactions on Knowledge and Data Engineering 2(1)*, IEEE, March 1990.

M. Stonebraker, A. Jhingran, J. Goh, e S. Potamianos, “On Rules, Procedures, Caching and Views in Database Systems[10]”, Proc. ACM-SIGMOD Conference on Management of Data, June 1990.

3. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M87-13.pdf
4. http://simon.cs.cornell.edu/home/praveen/papers/partindex.de95.ps.Z
5. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M85-95.pdf
6. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M87-06.pdf
7. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-82.pdf
8. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf
9. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M90-34.pdf
10. http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M90-36.pdf

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X